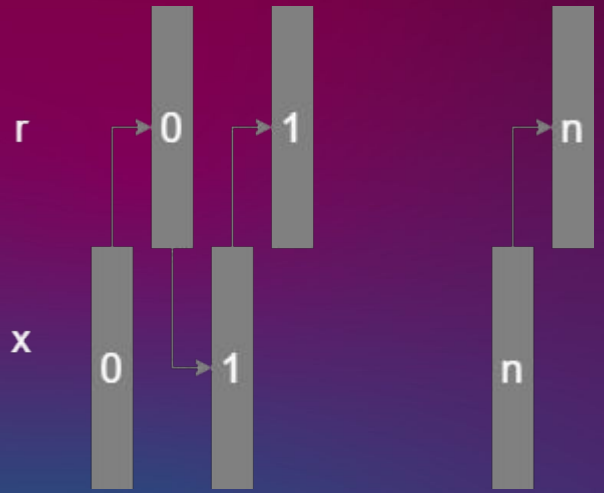
Implementing Asynchronous Jacobi Iteration on GPUs

Yu-Hsiang Mike Tsai[1], Pratik Nayak[1], Edmond Chow[2], Hartwig Anzt[3,1]

1. Steinbuch Centre for Computing, Karlsruhe Institute of Technology
2. School of Computational Science and Engineering, Georgia Institute of Technology
3. Innovative Computing Laboratory, University of Tennessee

Jacobi Iteration

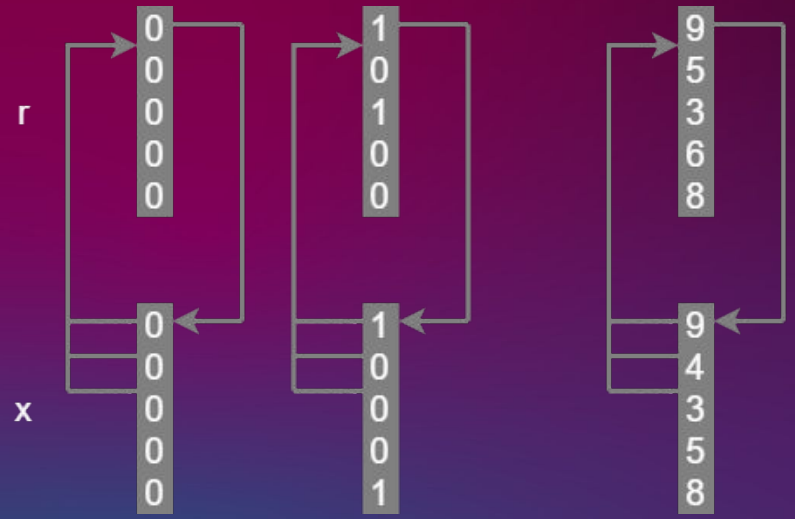
step by step



Algorithm 1 Synchronous Jacobi iteration

```
1: for i = 0 .. max_iterations-1 do
2:     Compute Residual: r = b - Ax
3:     Synchronize
4:     Update: x += α * r
5:     Synchronize
6: end for
```

Asynchronous Jacobi Iteration (static)

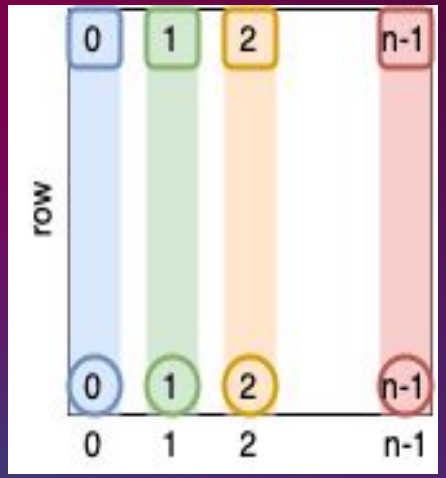


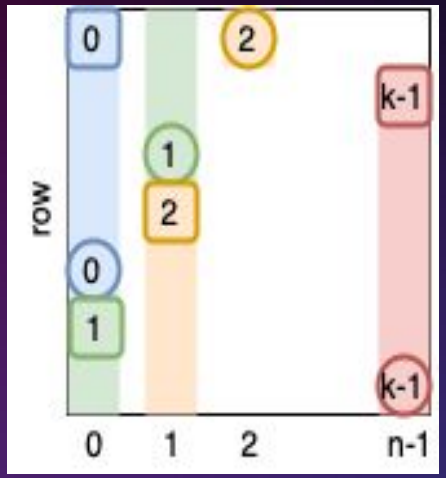
Algorithm 2 Static: assign subwarp to the same row

```
1: for all row in [0, #rows) in parallel do
2:     Accumulate: temp = A(row, :) * x
3:     Update: x[row] += α * (b[row]-temp)
4:     Enforce memory updated: __threadfence();
5: end for
```

Asynchronous Jacobi Iteration (dynamic)

Only assign the number of threads <= the number of rows based on the architecture



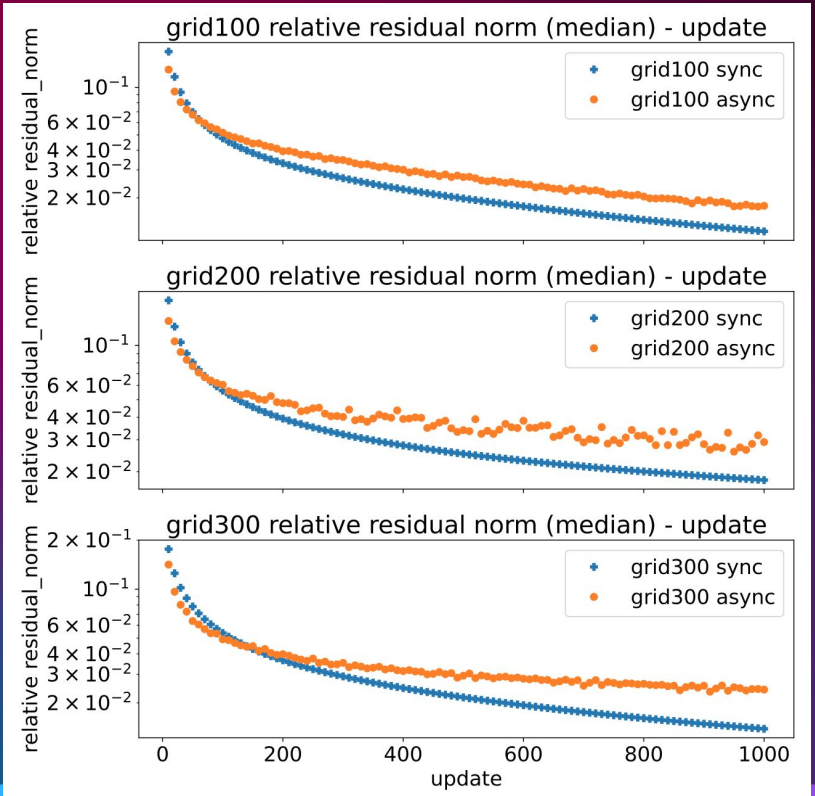


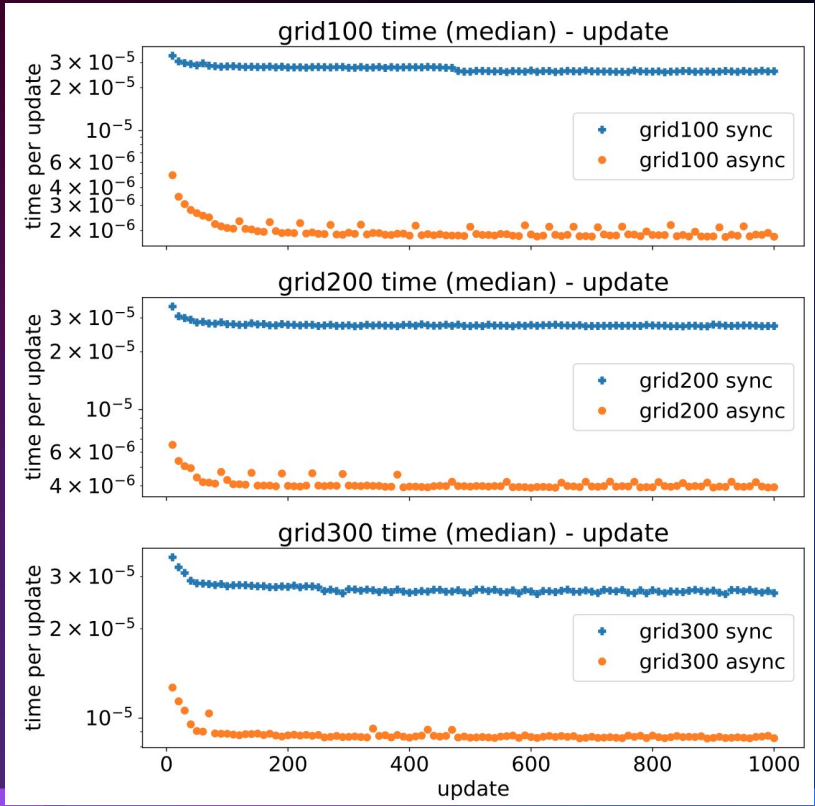
The same background means the same #update of thread.
They do not run simultaneously

Experiments setup

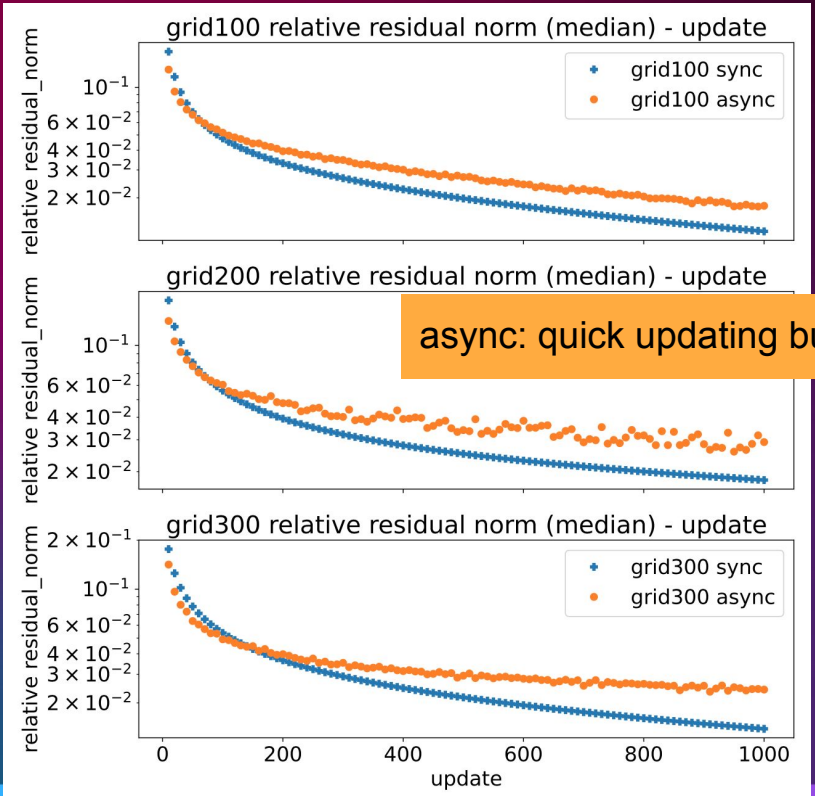
- We use CUDA 11.4.2 and g++ 9.3.0 (c++14) on V100 GPU
- Focus on Laplacin 2D 5pt stencil problem on grid sizes (100 x 100), (200 x 200), (300 x 300), which gives us matrices of size (10000 x 10000), (40000 x 40000), (90000 x 90000)
- The values are stored in double precision (IEEE 64-bit representation)
- We perform 10 warm-up runs and 100 runs for measurements
- The number of iterations is chosen from 10 to 1000
- Measurements are collected only after all iterations have completed.

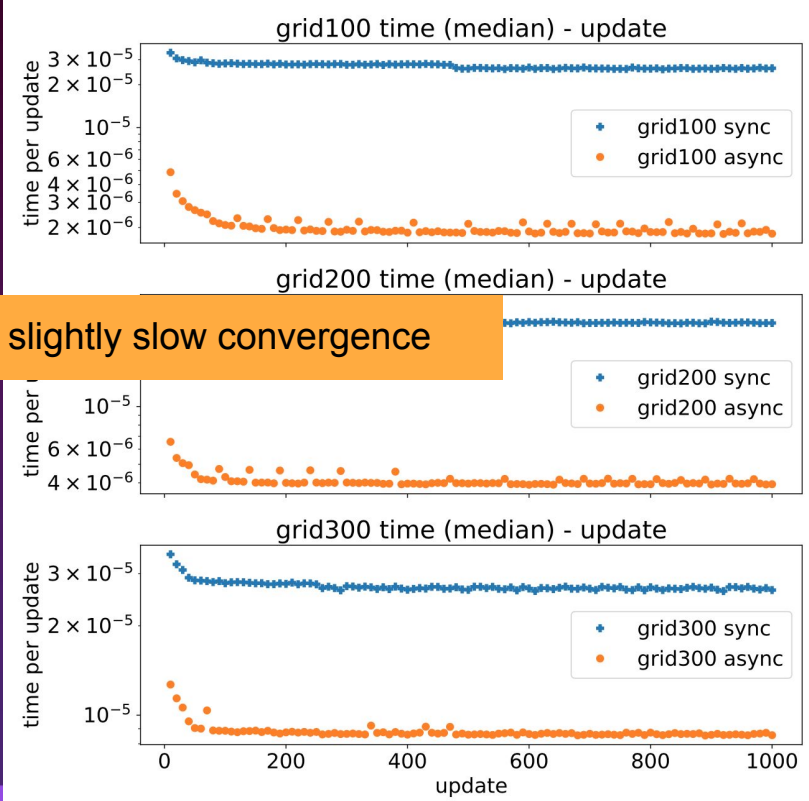
2D 5pt stencil problem - results per update





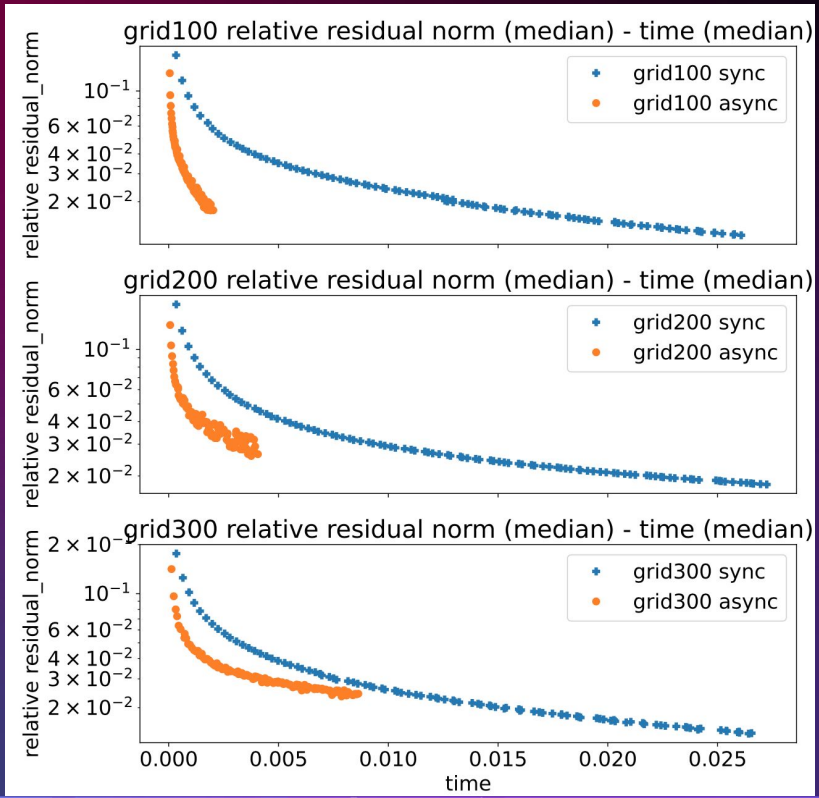
2D 5pt stencil problem - results per update



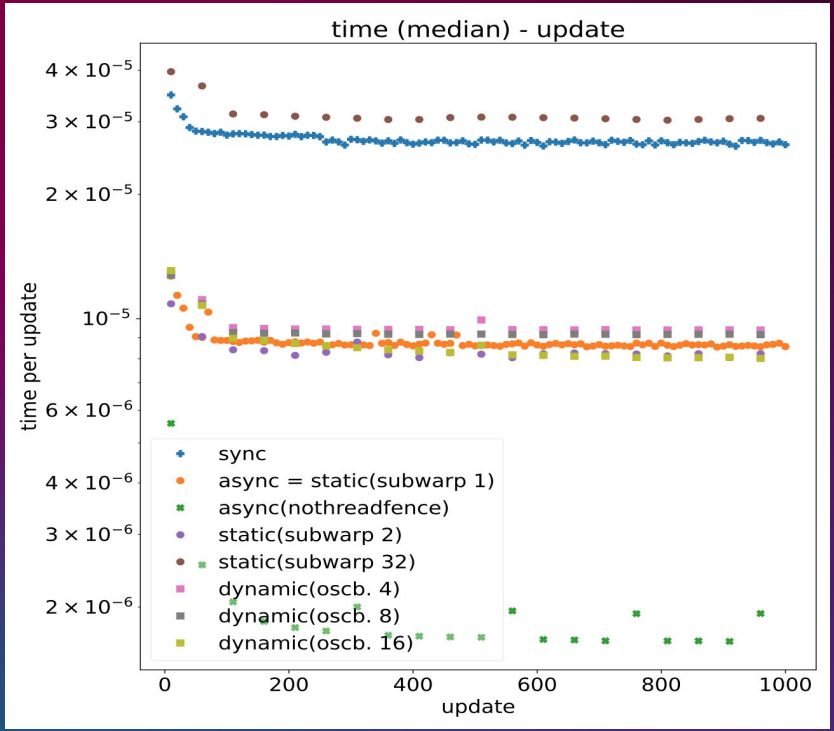


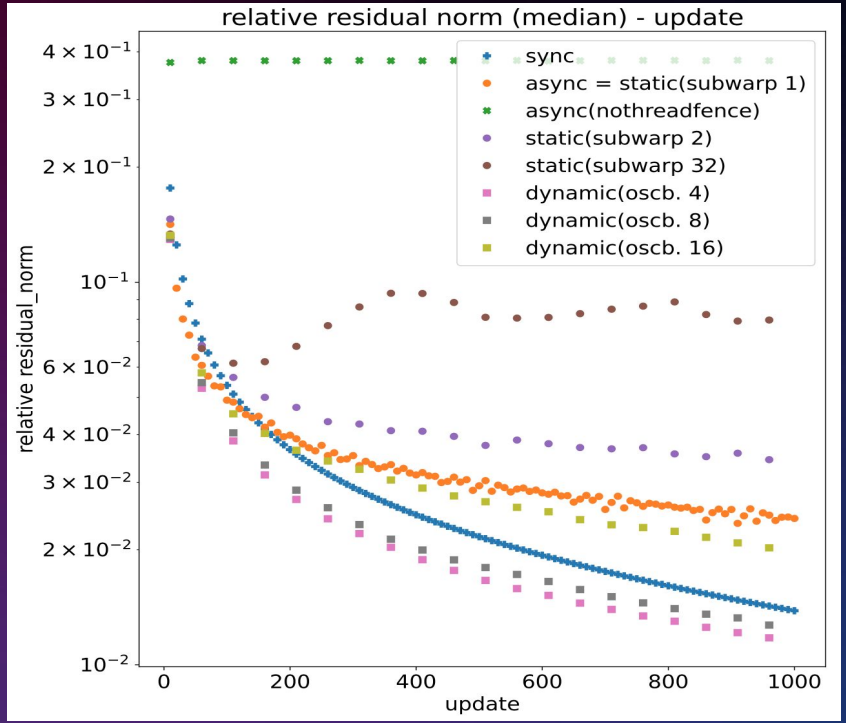
async: quick updating but slightly slow convergence

2D 5pt stencil problem - residual reduction over time

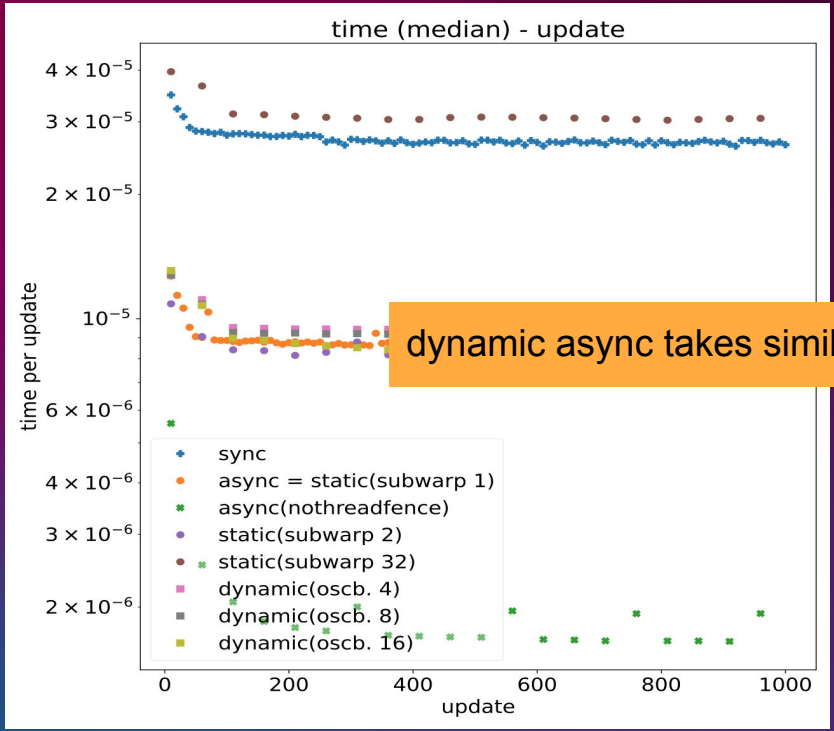


grid size (300 x 300) - results per update

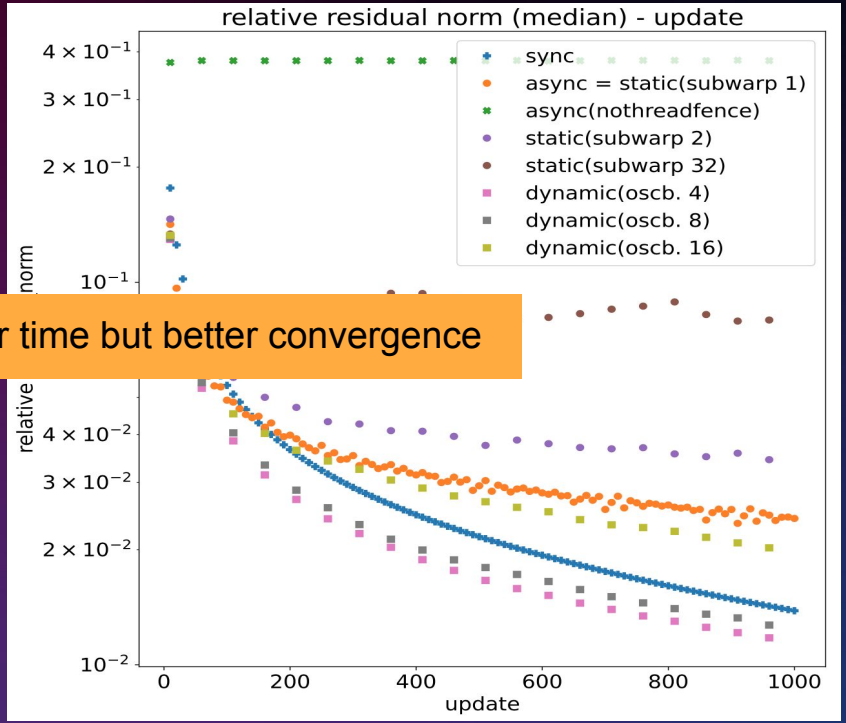




grid size (300 x 300) - results per update



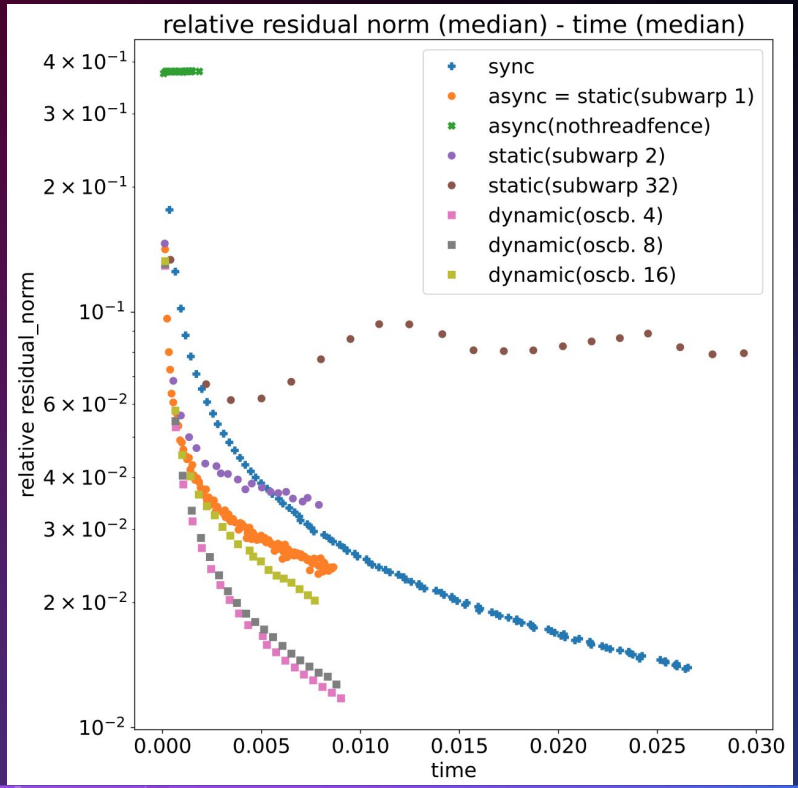
dynamic async takes similar time but better convergence



grid size (300 x 300) - residual norm reduction over time

Two kinds gives a different trend

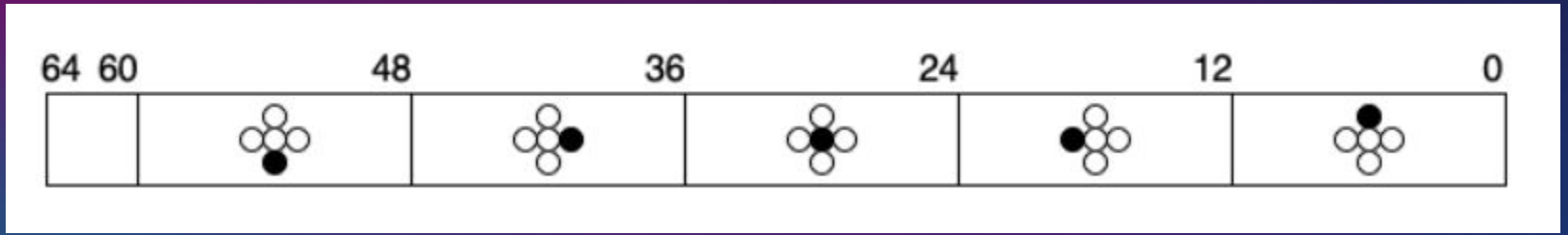
- async (nothreadfence)
- static(subwarp 32)



Analysis process

We focus on the 5-pt stencil stored in double precision.
We store the information to the output.
i.e. 5 information needs to be embedded in 64 bits. (64/5 ~= 12.~)

The value will be [0, 1, 2, 3, 4] not [-0.25, -0.25, 1, -0.25, -0.25].
With val * 12 shift and 0xFFF mask, we can extract/embed the information



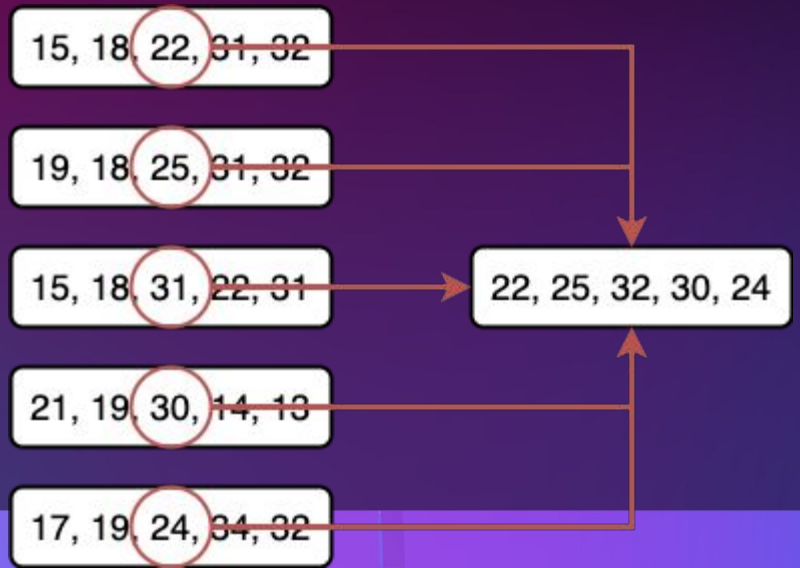
The recorded event

To avoid we are far from the practical implementation, we use alpha = 1 to ensure we have the same amount of memory read. i.e. update value with the desired information + (1 - alpha) * (the missing entry in the desired information)

- time: when does the whole process of each elements start and end ?
- update source: what is the index of source for this update ?
  - Final update value age: get the update source in the final update
  - Midway update value age: get the update source in the midway through whole process

The recorded event

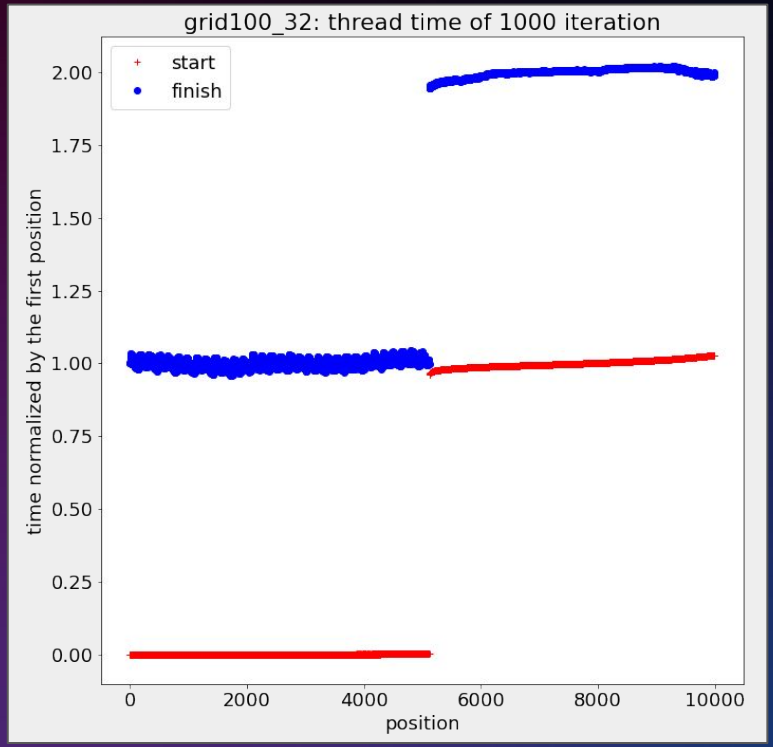
- time: when does the whole process of each elements start and end ?
- update source: what is the index of source for this update ?
  - Final update value age: get the update source in the final update
  - Midway update value age: get the update source in the midway through whole process



Too many threads leads update part by part

When the number of threads exceeds the number of parallel resource, cuda will finish the first part completely and then next part.

the limit of this problem on V100 is 80(#stream multiprocessor)*2048/32 = 5120

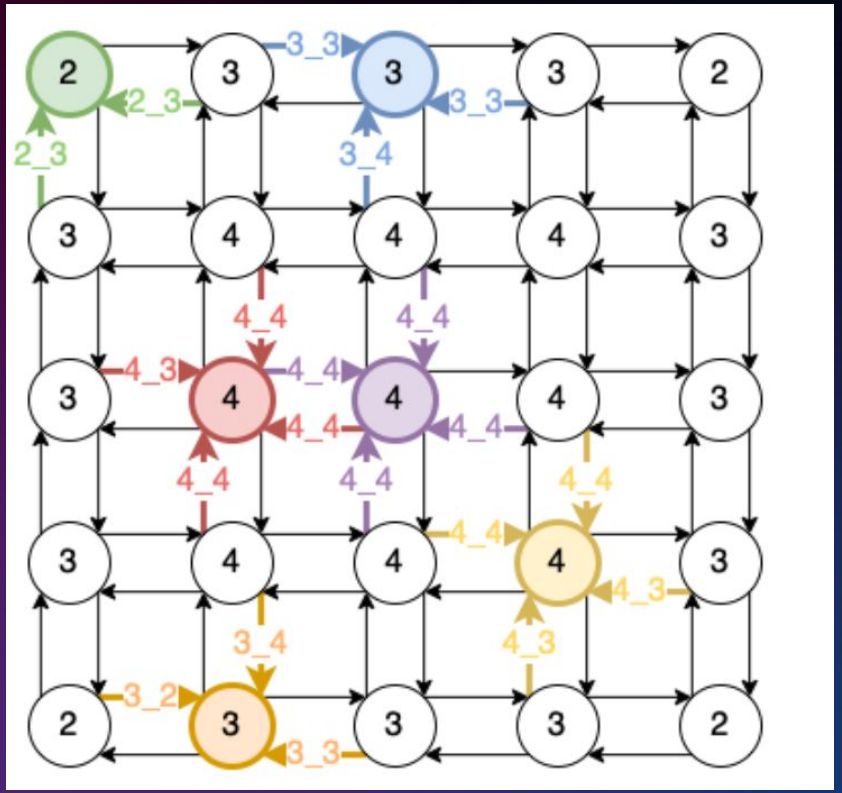


Group the update type

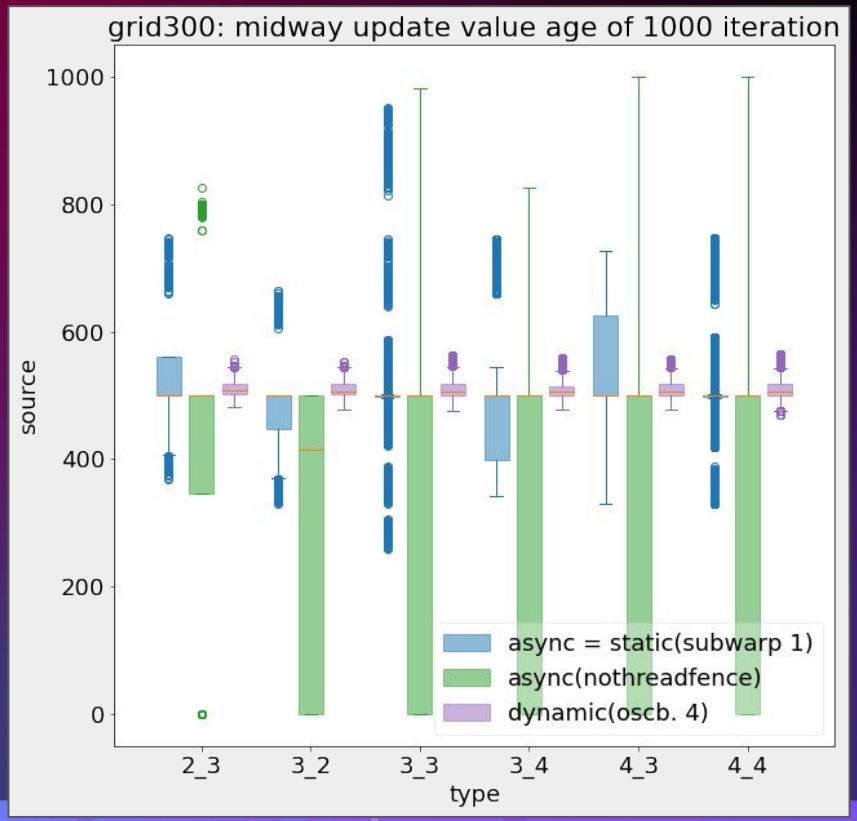
Each elements denotes the number of connection.

The update type can be represented as

<#connection of target>_<#connection of source>



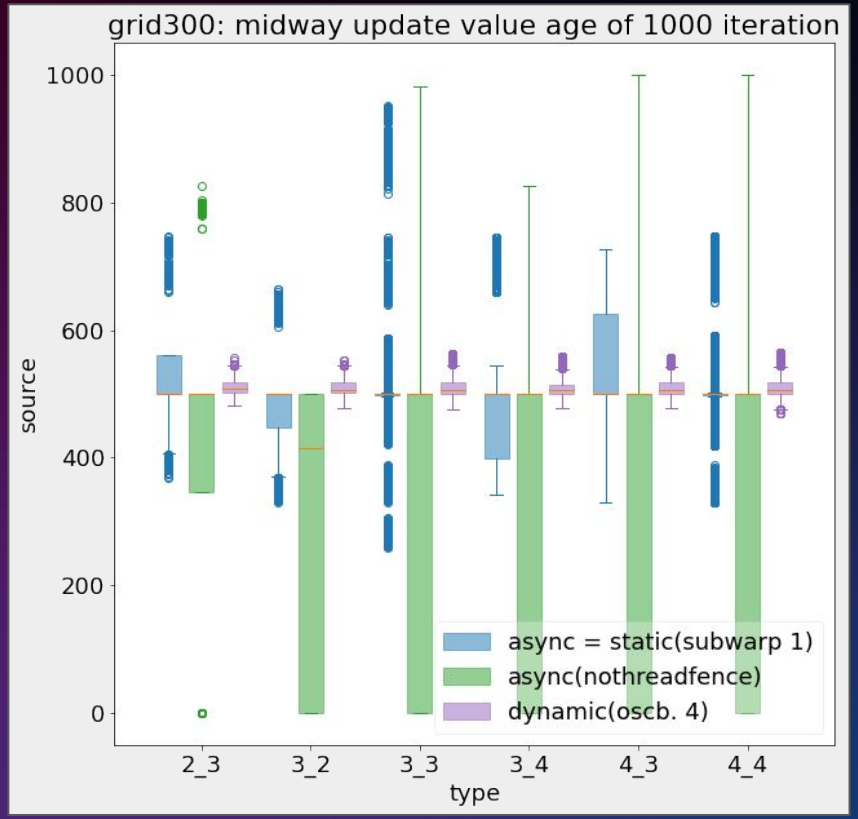
the boxplot of midway update value age of 1000 iteration



threadfence ensure update

threadfence is to avoid the outdated cache such that the order of update is indeed recognized by others. Without threadfence, the thread may take the old one in cache even the element is already updated in global memory.

Note. syncthreads/syncwarp gives a barrier among block/warp.

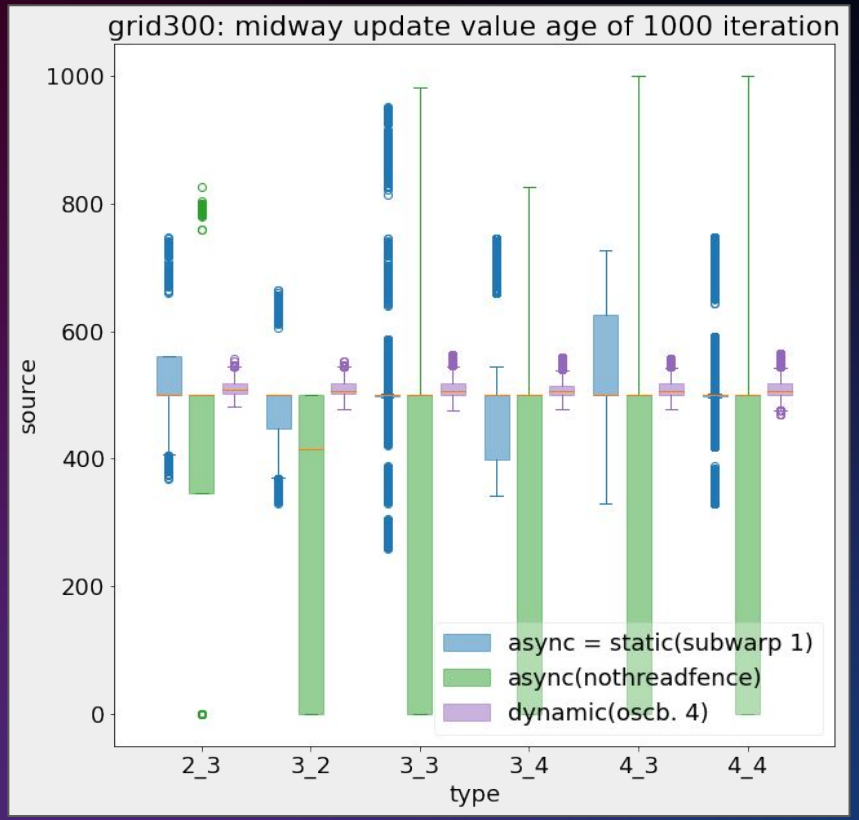


async static(subwarp 1) analysis

3_3 and 4_4 shows small variance in the boxplot due to the same work from the source to target.

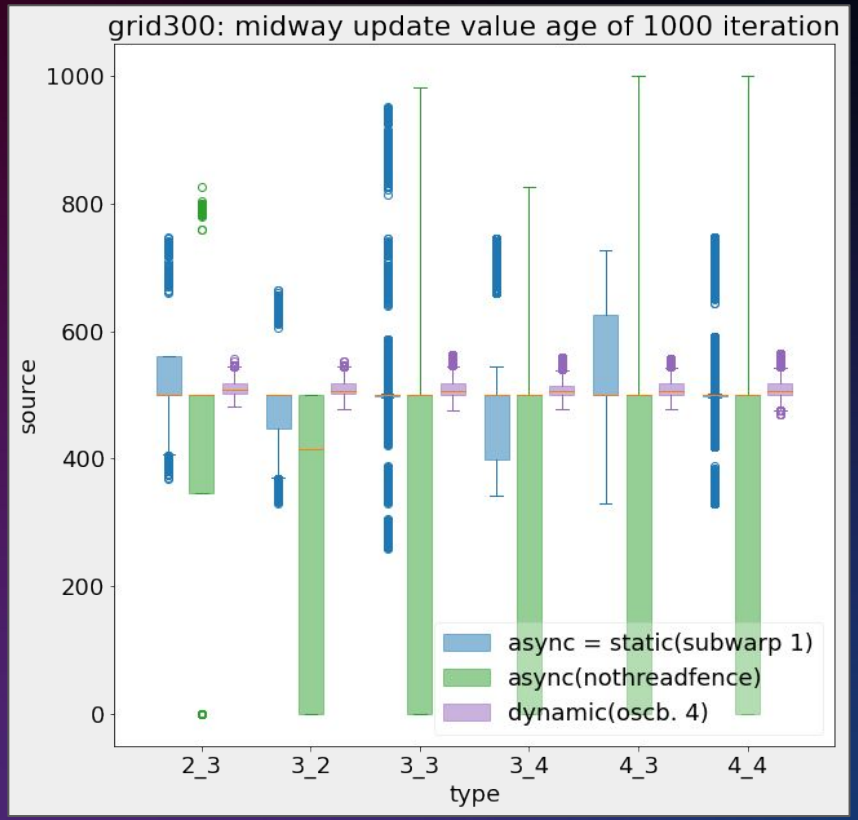
2_3/3_2 and 3_4/4_3 show the reversed trend

points with 3 connection has less work than points 4 connection such that 4(target)_3(source) shows the index of source is newer than the target

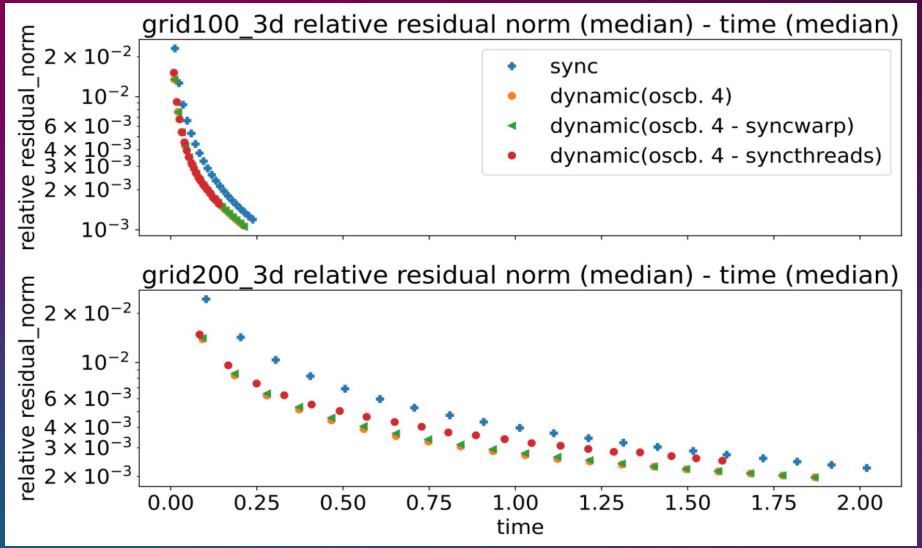


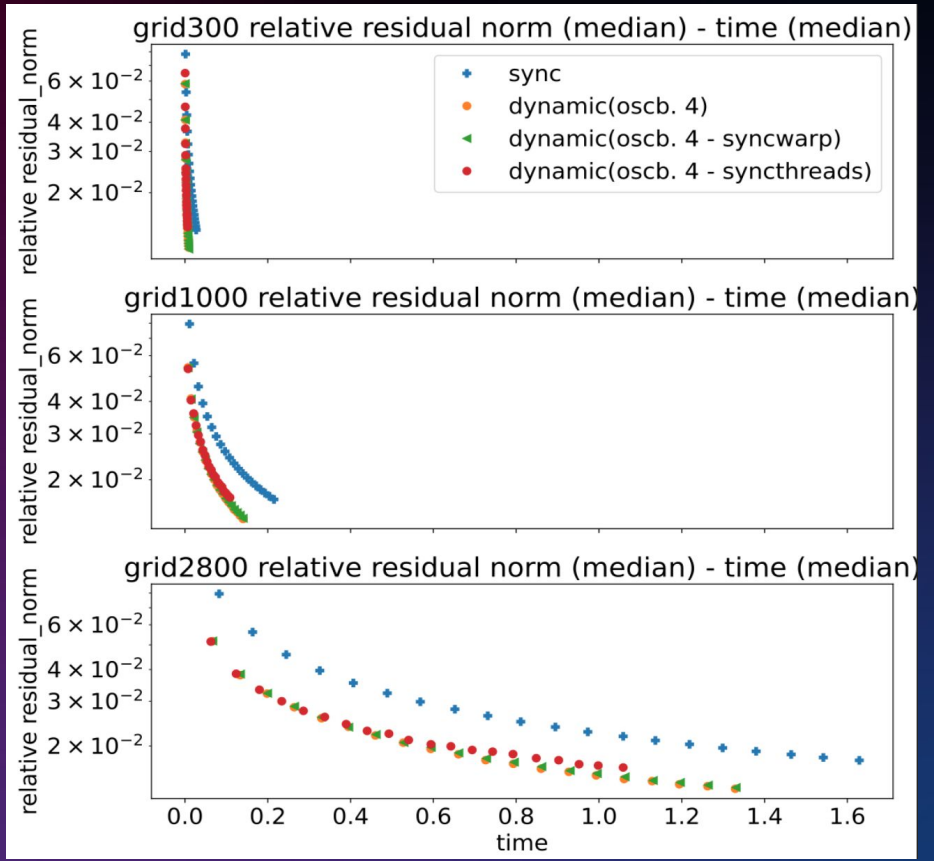
dynamic shows the smaller variance than static

The threads do not work on the same row, so it somehow shuffle the workload to give load balance.



Larger 2D and 3D cases





Conclusion

- We show the performance benefit from the asynchronous version
- We design a way to record information during the update such that we can analysis the reason of performance and the result quality
- It also works on the large case and the matrix with more dependence

The codes are available in https://doi.org/10.5281/zenodo.7130225 for these experiments, which based on Ginkgo sparse linear algebra library https://github.com/ginkgo-project/ginkgo
The numerical analysis is available in
Chow, Edmond, Andreas Frommer, and Daniel B. Szyld. "Asynchronous Richardson iterations: theory and practice." Numerical Algorithms 87.4 (2021): 1635-1651.